



freeBSDTM **JOURNAL**

Jan/Feb 2016

Building
Distributed
Applications

with

Apache Zookeeper

ALSO

The **DOS & DON'TS**
of **FILE SYSTEM**
BENCHMARKING

Using **FUZZY TESTING**
to Build **INDUSTRIAL**
STRENGTH SYSTEMS

Introducing the new **XG-2758 1U pfSense® Security Appliance**



Fast 10 Gigabit networking at a price you can afford.

XG-2758 1U features include:

- 8 Core Intel® Atom™ C2758 2.4 GHz with AES-NI and Quick Assist Technology.
- 16GB ECC RAM
- 4x 1Gb Ethernet RJ45 via Intel i354 on-chip; 1 port configurable RJ45 or SFP.
- 2x 10Gb Ethernet SFP+ via Intel 82599 Niantic.
- Optional PCIe x8 slot available for further expansion.
- Preloaded with pfSense Open Source software. No maintenance, licensing or upgrade fees.
- Flexible Configuration - firewall, LAN or WAN router, VPN appliance, DHCP Server, DNS Server, multi-WAN and high availability.
- Fully extendable with add-on software packages such as Snort®, Squid, SquidGuard, Suricata, to enable IDS/IPS, load balancing, traffic optimization, reporting and monitoring.
- Create VPNs to the Amazon Cloud easily with our with Amazon® AWS™ Wizard.



Shop now at the official pfSense store or authorized partners worldwide.

<http://store.pfsense.org/XG-2758>

pfSense® is a registered trademark of Electric Sheep Fencing, LLC. Intel and Intel Atom are trademarks of Intel Corporation in the U.S. and/or other countries. Amazon AWS and Amazon are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries. Snort is a registered trademark of CISCO.



FreeBSDTM JOURNAL Jan/Feb 2016

Table of Contents

Building Distributed Applications with **APACHE ZOOKEEPER** 4

On the surface, the problem of how to create a system that allows for reliable distributed communication and coordination can appear to be fairly trivial. While it may become tempting to just roll your own solution, you will find no shortage of horror stories that will quickly sour you on that idea. **By Steven Kreuzer**

Columns & Departments

3 Foundation Letter

An Exciting Year Ahead!
By George Neville-Neil

27 Ports Report The level of activity during the November–December 2015 period was especially high on the problem-fixing front. *By Frederic Culot*

28 svn update It's the start of a brand new year, and there have been several exciting updates that highlight FreeBSD's continued commitment to high performance networking. *By Steven Kreuzer*

30 This Month in FreeBSD

Over the next few issues, we'll take a closer look at some of the new features making their way into the 2016 releases and the developers behind those features. This month we chat with Allan Jude. *By Dru Lavigne*

34 Book Review A Review of *FreeBSD Device Drivers: A Guide for the Intrepid* by Joseph Kong. *By Simon Gerraty*

35 Events Calendar

By Dru Lavigne

Using Fuzzy Testing to Build Industrial-Strength Systems

One tool to improve the quality of the FreeBSD kernel is the stress2 stress-test suite, which can help you expose design and implementation problems while adding to or updating the kernel code. **By Peter Holm**



The Dos & Don'ts of File System Benchmarking

The file system is the cornerstone of any Unix derivative, and its performance and efficiency are extremely important for overall system speed. **By Vasily Tarasov, Zhen Cao, Ming Chen, and Erez Zadok**



#MISSIONCOMPLETE



"CryptoLocker is a joke with ZFS"

Learn how Plextec defeats ransomware attacks with FreeNAS and ZFS at ixsystems.com/cryptolocker

Have you used one of these tools to complete your mission?

Tell us more at ixsystems.com/missioncomplete for a chance to win monthly



#missioncomplete





John Baldwin • Member of the
FreeBSD Core Team

Justin Gibbs • Founder and President of the
FreeBSD Foundation and a
senior software architect at
Spectra Logic Corporation

Daichi Goto • Director at BSD Consulting Inc.
(Tokyo)

Joseph Kong • Author of *FreeBSD Device
Drivers*

Dru Lavigne • Director of the FreeBSD
Foundation and Chair of the
BSD Certification Group

Michael W. Lucas • Author of *Absolute FreeBSD*

Kirk McKusick • Director of the FreeBSD
Foundation and lead author of
The Design and Implementation
book series

George Neville-Neil • Director of the FreeBSD
Foundation and co-author of
The Design and Implementation
of the *FreeBSD Operating System*

Hiroki Sato • Director of the FreeBSD
Foundation, Chair of
AsiaBSDCon, member of the
FreeBSD Core Team and
Assistant Professor at Tokyo
Institute of Technology

Robert Watson • Director of the FreeBSD
Foundation, Founder of the
TrustedBSD Project and
Lecturer at the University of
Cambridge

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Art Director • Dianne M. Kischitz
dianne@freebsdjournal.com

Office Administrator • Michael Davis
davism@freebsdjournal.com

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0)
is published 6 times a year (January/February,
March/April, May/June, July/August,
September/October, November/December).

Published by the FreeBSD Foundation,
PO Box 20247, Boulder, CO 80308
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdjournal.org
Copyright © 2016 by FreeBSD Foundation.
All rights reserved.

This magazine may not be reproduced in whole or in
part without written permission from the publisher.

An Exciting Year Ahead!

With the new year and this issue, we say hello to a new columnist at the *Journal*. Steven Kreuzer takes over the svn update column from Glen Barber, who, with the advent of FreeBSD 11.0, needs more time for his release engineering tasks. It has been great having Glen on board, we thank him for two years of great columns, and we're sure that Steven will do a wonderful job as the torch is passed to him.

This first issue of 2016 features three excellent articles. The Dos and Don'ts of File System Benchmarking treats a topic that isn't covered often enough. While many people think they know how to measure system performance, most are wrong. A cornerstone article on file system benchmarking that can educate the broader technical community is one of our topmost goals.

Security researchers know and apply fuzz testing to systems to find security bugs, but this method can be applied more broadly and has found acceptance as a general tool for finding bugs and improving systems. Peter Holm provides a very personal overview of a tool he's built and applied to the FreeBSD kernel, but which is actually applicable to software in general.

Rounding out the articles this month, you'll enjoy reading a piece that looks at one of the more popular distributed coordination systems, Zookeeper. The author, Steven Kreuzer, not only uses Zookeeper, but also maintains the port for the FreeBSD ports system.

In 2016 *Journal* will feature interviews with people from the FreeBSD community. Dru Lavigne, a member of the *Journal's* Editorial Board and the author of the *Journal's* history and calendar columns, interviews Allan Jude, who has recently co-authored a book on ZFS. Over the course of the year, we'll present interviews with more community members to learn about what brought them to FreeBSD and what they get from and give back to the project.

And don't overlook this issue's informative columns, one of which is a new book review, by Simon Gerraty, of *FreeBSD Device Drivers: A Guide for the Intrepid*, authored by Joseph Kong.

Lastly, we don't generally put spoilers in the *Journal*, but it's too exciting not to alert readers that later in the year we will dedicate a full issue to FreeBSD 11, the latest release of FreeBSD planned to ship this summer. We won't give away all the articles the special issue will include, but we do want everyone to watch for both the new release and, of course, the *Journal* issue that will follow.

Sincerely,
George Neville-Neil

For the *FreeBSD Journal* Editorial Board

Building Distributed Applications with

Apache Zookeeper

By Steven
Kreuzer

Distributed systems are very complex beasts. While each system is built to solve a unique problem, they all share a common requirement of having a way for all nodes to communicate with each other in a reliable, fault-tolerant, and scalable fashion. On the surface, the problem of how to create a system that allows for reliable distributed communication and coordination appears to be fairly trivial, and you will find no shortage of academic papers that describe these algorithms in great detail. After a while, it may become tempting to just roll your own solution, but ask anyone who has done this before and I am sure you will find no shortage of horror stories that will quickly sour you on that idea. Since it is safe to assume that your time will be better spent not having to debug subtle race conditions and deadlocks, one could make a strong argument that you will be better served by deploying an existing solution that has already seen widespread adoption by large, open-source projects, universities, and companies across all types of industries. One such piece of software is Zookeeper, which is an Apache project focused on building a robust system to implement distributed system primitives that developers can immediately put to use when writing a distributed application. Originally created at Yahoo!, it is now an actively developed and vital component of the Hadoop ecosystem. While many Hadoop-related projects make extensive use of Zookeeper, aside from Java, it has no outside dependencies, which makes it incredibly simple to introduce it into your environment.

The Zookeeper Data Model

At its core, Zookeeper provides just a few basic operations that can be used to form many of the design patterns that are necessary in a distributed system. Zookeeper allows applications to communicate and coordinate with each other through hierarchical, key-value stores referred to as zNodes. The interface is done in such a way that it closely resembles a typical UNIX file system, with each zNode acting as either a file that is able to store up to a megabyte of data or as a directory that can contain multiple child zNodes. In addi-

tion, each zNode has some additional metadata such as the ACLs, creation time, modification time, and a version number associated with it.

Zookeeper allows for the creation of two different types of zNodes: persistent and ephemeral. When a new zNode is created, it will be persistent by default. Just as the name implies, the zNode will remain available until it is explicitly removed through the delete function. An ephemeral node will only persist while the client who created the zNode is connected. If the client's session is disconnected due to either a crash or explicit termination, Zookeeper will remove the node. Ephemeral nodes are very useful for providing host and service discovery and can also be used as a simple way to detect faults in a distributed system. While ephemeral zNodes can be created as a child of persistent zNodes, an ephemeral zNode does not have the ability to have child zNodes. In Zookeeper, an ephemeral zNode will always be a leaf node.

Zookeeper offers two additional components that, when combined with zNodes, allow you to easily replicate very complex behavior in a straightforward fashion. Both persistent or ephemeral nodes can be part of an atomic sequence whose names are automatically assigned a monotonic number which is maintained by the parent zNode. Zookeeper guarantees that this 10-digit number will always be unique and greater than any other child zNode created under the parent zNode. These sequential zNodes can be used as building blocks to easily implement a distributed locking mechanism if your application has such a requirement. Zookeeper also features the concept of a watch, which allows a client to request that it be notified when a change is made to a specific zNode. Watches can be used as simple mechanisms to create asynchronous, event-based systems and easily implement leader election algorithms. A watch in Zookeeper is a one-time trigger. After a change occurs and the notification is sent, the client must once again register the watch to receive notifications of future updates.

Getting Up and Running

Zookeeper was added to the FreeBSD ports tree in 2012, which makes it incredibly simple to get up and running with a single instance for testing and development. Because Zookeeper was defined to work out of the box and requires very little configuration to get going, it should not be necessary for you to have to make any changes to the config file until you are ready to deploy into your production environment.

```
# pkg install zookeeper
# sysrc zookeeper_enable=YES
zookeeper_enable: -> YES
# cp /usr/local/etc/zookeeper/zoo_sample.cfg /usr/local/etc/zookeeper/zoo.cfg
# service zookeeper start
Starting zookeeper.
```

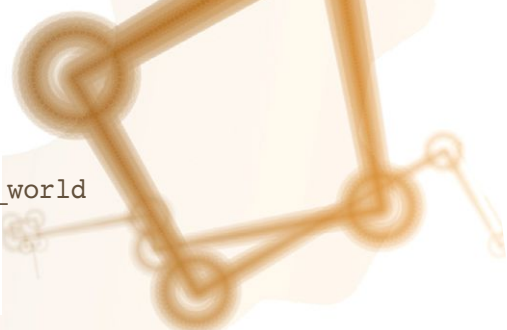
Once the server is up and running, you can connect to the server using the `zkCli.sh` command and try out a few commands to verify that everything is working as expected.

```
$ zkCli.sh
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is enabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0] ls /
[zookeeper]
```

Let's first start off by creating a new zNode called "test" and populate it with the data "hello_word." After that is done, we can list the contents of the root zNode and see that the new "test" zNode exists.



```
[zk: localhost:2181(CONNECTED) 1] create /test hello_world
Created /test
[zk: localhost:2181(CONNECTED) 2] ls /
[test, zookeeper]
```

The contents of the “test” zNode can be retrieved by issuing a get on the zNode. In this example the string “hello_world” along with some additional metadata about the zNode itself will be returned.

```
[zk: localhost:2181(CONNECTED) 3] get /test
hello_world
cZxid = 0x6
ctime = Tue Dec 29 15:39:07 GMT 2015
mZxid = 0x6
mtime = Tue Dec 29 15:39:07 GMT 2015
pZxid = 0x6
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 11
numChildren = 0
```

The contents of the “test” zNode can also be updated using the “set” command. You will notice that after the command is run, metadata such as “mtime,” “dataVersion” will also automatically be updated as well. Once the update has completed, another “get” can be issued to verify that the contents of the zNode have been modified.

```
[zk: localhost:2181(CONNECTED) 4] set /test freebsd_journal
cZxid = 0x6
ctime = Tue Dec 29 15:39:07 GMT 2015
mZxid = 0x7
mtime = Tue Dec 29 15:49:46 GMT 2015
pZxid = 0x6
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 15
numChildren = 0
```

```
[zk: localhost:2181(CONNECTED) 5] get /test
freebsd_journal
cZxid = 0x6
ctime = Tue Dec 29 15:39:07 GMT 2015
mZxid = 0x7
mtime = Tue Dec 29 15:49:46 GMT 2015
pZxid = 0x6
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 15
numChildren = 0
```

Once this zNode is no longer needed, a “delete” can be issued to completely remove the zNode from the data store.

```
[zk: localhost:2181(CONNECTED) 6] delete /test
[zk: localhost:2181(CONNECTED) 7] ls /
[zookeeper]
```

Moving into Production

While having a single instance of Zookeeper running in stand-alone mode is perfectly fine to use for testing or to do some quick prototyping, it does introduce a single point of failure that should be addressed before mission critical applications start to depend on it. Once you are ready to introduce Zookeeper into a production environment, you will want to deploy it configured to run in what's referred to as quorum mode. While Zookeeper is used for coordinating distributed applications, it, too, is a distributed application in which independent machines can form a cluster and elect a leader. The cluster, which is referred to as an ensemble, will replicate its data to all members, and as long as a majority of the nodes in the ensemble are online, all the services that Zookeeper provided will be available.

When researching the requirements for an ensemble, it is recommended that you start with at least three machines, but for most environments it is strongly encouraged to have at least five machines at a minimum. The reason for this is that in a three-node cluster, the loss of a single node is tolerable because two of the three remaining machines will still count as a majority. However, let's say you remove a node from the ensemble for regular scheduled maintenance and during this time you were to have another node unexpectedly fail. Now that the quorum is lost, the remaining nodes will switch to peer election mode and will disconnect existing clients and refuse new connections until a new leader has been elected. This particular failure scenario can be avoided by starting with a minimum of five nodes, which will allow the cluster to tolerate the loss of up to two nodes. While it is possible to configure Zookeeper to operate in read-only mode if the quorum is lost, determining if this feature is appropriate for your environment will mainly be driven by the requirements of your application, and as a result, the default behavior is to simply stop serving client connections.


Since Zookeeper's main focus is to ensure that data is distributed in a reliable manner, another recommendation when deploying an ensemble is to always have an odd number of machines. This prevents the possibility of having a "split-brain" in which certain nodes become segmented from the other nodes but continue to operate independently. In this scenario, these machines can fall out of sync with the other half and once the failure has been resolved, the cluster won't know how to reconcile the differences. To combat this, Zookeeper uses a majority count, and a new node will be selected as the leader.

At the heart of Zookeeper is an atomic messaging system designed to keep each member of the ensemble in sync. The node that is elected as a leader will receive all writes and is responsible for publishing those changes to all the other members acting as a follower of the leader node. Zookeeper guarantees that data will eventually be consistent across all members of the ensemble by ensuring that data is always delivered in the same order it is sent. A message will only be delivered after all messages sent before it have been delivered, and while it's possible that two clients may not have the exact same point in time view, they will always observe the changes in the same order.

In quorum mode, all nodes have a copy of the same configuration file and will know about every other machine that is a member of the ensemble. This is accomplished by appending additional lines into the zoo.cfg file in the form of **server.id=host:quorum_port:election_port**.

```
# cat /usr/local/etc/zookeeper/zoo.cfg
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/db/zookeeper
clientPort=2181
server.1=zook1:2888:3888
server.2=zook2:2888:3888
server.3=zook3:2888:3888
```

Within the ensemble, each node must have a unique id between 1 and 255 assigned to it. The node is informed of its id by reading the contents of the "myid" file stored in the directory, which is



defined by the `dataDir` directive in `zoo.cfg`. The file consists of a single line containing only the text of that machine's id. For example, the id of the node named `zook2` is defined as 2 in the configuration file. On that node you can create the `myid` file using the command `echo 2 > /var/db/zookeeper/myid`. Once this has been done on each node in the ensemble, you can simply start up Zookeeper the same way you did when it was configured in stand-alone mode. Each node in the ensemble will reach out to one another to run an election to select a leader with all the other nodes becoming followers.

One of the big advantages is that it's not much work at all to go from stand-alone mode on a single machine to the highly fault-tolerant quorum mode distributed across multiple machines in your environment. What's even better is that from a developer's standpoint, all the interfaces are still the same, and so no additional changes to the application will be necessary. One of the most appealing features of Zookeeper is that the system is designed to require little maintenance after the initial setup, and all the complexity is hidden away from the end user, which makes it easy to integrate.

Scaling Zookeeper


Normally within a distributed system, you can scale up to handle an increase in workload by adding additional capacity and spreading your application across more nodes. However, because each member of the ensemble needs to reach agreement on every transaction, as you add more voting members, the write performance of the ensemble will start to decrease. In addition to leader and follower roles, Zookeeper also allows members to join the ensemble and act as observers. When assigned this role, an observer will not take part in the agreement step of the atomic broadcast protocol. Instead, it will just accept transactions that have already been agreed upon by other followers in the quorum. The main goal of an observer is to provide read scalability without having to compromise on write performance. In addition, because observers do not vote or participate in leader elections, they can be more heavily loaded with read-only clients, which allows you to reduce the load placed on follower nodes that are required to participate.

Configuring a node to act as an observer is a fairly straightforward procedure. The node needs to be informed that it should act as an observer by adding the line `peerType=observer` into the configuration file for that particular node. In addition, all the other nodes are informed which servers are acting as observers by appending `:observer` at the end of the server config line.

```
server.1=zook1:2888:3888
server.2=zook2:2888:3888
server.3=zook3:2888:3888
server.4=zook4:2888:3888:observer
```

Once these configuration changes are in place you can restart the cluster normally and clients will be able to connect to an observer node in exactly the same way they would if it was a follower node.

Conclusion



When done incorrectly, a distributed system can quickly become an unmanageable mess that is hard to use and even more difficult to debug. As datasets become larger and workload demands become more intense we are quickly approaching a point in time when to get more work done, your only option will be to distribute the processing to as many machines on the network as possible. Because of this shift in technology, it comes as no surprise that Zookeeper has quickly become such a popular open-source project and has gained such widespread adoption. If you ever find yourself needing to introduce complex functionality into your application, Zookeeper will quickly become an important piece of infrastructure in your software stack. ●

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.



Your donations over the past year have led to great progress for FreeBSD and the community. **Thank you!!**

We look forward to continuing that progress as we head into 2016. Your investment will help in the following areas:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Thank you for all of your continued support.

You make FreeBSD possible!

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Making a donation is quick and easy.
Go to freebsd.foundation.org/donate



The
FreeBSD
FOUNDATION
freebsd.foundation.org

USING FUZZY TESTING TO BUILD

Industrial-Strength Systems

One tool to improve the quality of the FreeBSD kernel is the stress2 stress-test suite, which can help you expose design and implementation problems while adding to or updating the kernel code. It was first checked into svn in 2009, but had been in use for years before that. In 2014 more than 100 problem reports were generated using it.



By Peter Holm

How I Got Started with Stress Testing

Sometime during the '90s I was asked by the support group in the company I worked for to look at a customer's computer that would crash every time it ran a vendor database application. To my amazement, the computer panicked in a frequently used syscall. Stepping through the code, I realized that it was a simple stack-trashing bug in a user-mode program that had triggered the problem. Two things were forever imprinted on my mind: a) although it was interesting to meet the customer's top management, it would be much nicer to find these kinds of errors back at the office, and b) I was amazed that a common user-mode bug had crashed the computer! The same day, I wrote my first stress test program—syscall fuzzer—and that kept a lot of people busy for days.

Test Design

A widely used test in the FreeBSD Project is "buildworld." It is a good test, but when you have run it a few times in a row there is no gain in continuing. Stress2 uses a different strategy for most of the test cases, as tests run differently each time they are run—different runtime, different number of threads, and different VM pressure. The benefit of this strategy is that it provides greater coverage. The downside is that it is slightly harder to reproduce errors, but this problem is really negligible compared to the benefits.

Running the same test in multiple threads is used by many scenarios to stress locking of shared resources. Some tests synchronize the threads, but others will not, in the hope of achieving a wider test coverage. A cornerstone of Stress2 is that VM pressure is added to most tests to trigger waiting points in the kernel. A large number of tests have run without any detectable issues, but the moment VM pressure is added, panics and deadlocks show up.

A few tests rely on fuzzing. An example is the mmap(2) family of tests.

A Typical Workday with Stress2

I get a patch from one of the FreeBSD committers that is supposed to fix a panic. The problem may already be triggered by some of the existing test scenarios or a new test has to be written. Once the problem can be repeated within a reasonable time frame, it's time to test the patch. When the fix has been verified, I typically run all of the other scenarios, just in case the fix has some side effect, which happens surprisingly often. It is not unusual for a patch to go through a few revisions before commit. On a really productive day, I'm able to go through three or four revisions. I try to generate bug reports with enough information to ensure a fast patch update. A bug report typically looks like this:

<https://people.freebsd.org/~pho/stress/log/kostik833.txt>.

A full test usually takes more than 24 hours before I'm confident the fix is OK.

When working on a new test scenario, the by-product is quite often one or more new test cases that trigger other problems than the one reported. These are marked for commit, even though they may be WIP. The philosophy here is that any test that can crash the kernel is a good test.

As can be seen by this example, it is a team effort. I work with a bunch of extremely talented people, and as a team we are incredibly productive.

Memory Leaks

Memory leaks get introduced on a regular basis. I check this by watching `vmstat -m` and `-z`. There is a script I run during tests that does this automatically: `stress2/tools/vmstat.sh`

Test Hardware

I have observed that different types of hardware make a difference, as some problems can only be reproduced on one type. Disk speed also plays a role and so does the number of CPUs. Lately I was reminded of the importance of this. I had a patch for evaluation, which survived all tests. Others, however, found that a continuous `-j7 buildworld` on a 6-core machine configured with 1GB of RAM would trigger a panic. Having swap configured is mandatory for quite a few of the tests. Without swap, it is very hard to balance the right amount of pressure. That is, without a swap disk, too much VM pressure triggers OOM killing. I test both `i386` and `amd64` on real hardware and I use a serial console for all test hosts, so I can log the output.

Examples

The following describe a few test scenarios I found especially interesting. All are found in the `stress2/misc` directory. Common for all tests in this directory is that they are all complete and self-contained test scenarios, implemented as shell scripts. The majority are regression tests; that is, they have once triggered a panic or a deadlock. The tests can be run individually. Most, but not all, tests use a memory-based file system for the tests. The primary benefit is that file system is in an initial consistent state.

The following show what a test in `stress2/misc` can look like:

```
#!/bin/sh

# "panic: not suspended thread 0xc674c870" seen.

for i in `find /proc ! -type d`; do
    dd if=$i of=/dev/null > /dev/null 2>&1
    dd if=/dev/random of=$i > /dev/null 2>&1
done
```

Many of the tests in `stress2/misc` use the more general stress test programs found in `stress2/testcases`.

trim6.sh

Quite often test cases written for one purpose also catch different problems. This test case was originally written for a problem deleting a large file on a file system with option TRIM enabled, but was later able to trigger both a deadlock and a panic during file creation. The test in this example is quite

simple: write a very large file to a fast (SSD) disk. From the r287361 commit log: By doing file extension fast, it is possible to create excess supply of the D_NEWBLK kinds of dependencies (i.e., D_ALLOCDIRECT and D_ALLOCINDIR) which can exhaust kmem.

crossmp.sh

A different and productive example of parallelization is the Cross Mount Point test case, which is responsible for 6 bug reports. The tests mount and unmount 15 different file systems / mount points in parallel.

callout_reset_on.sh

The scenario I have spent the most time on is from pr. kern/166340, a very detailed bug report: Processes under FreeBSD 9.0 would hang in uninterruptible sleep with apparently no syscall (empty wchan). A later change to the callout wheel would trigger a panic: Bad link elm 0xfffff80012ba8ec8 prev->next != elm with this scenario.

mmap10.sh

This is one of the fuzz test cases. The general idea is to pass random values to system calls in an attempt to flush out errors in the code. Once you get past the simple missing parameter validation problems, more interesting problems tend to surface. This test scenario generated the following unique problems by passing random values to mlock(2), mprotect(2), and mlockall(2):

```
panic: deadlkres: possible deadlock detected for 0xcb0ea930, blocked for 1801709 ticks
panic: pmap active 0xfffff800a90cfd78
panic: vm_fault_copy_wired: page missing
panic: vm_object_backing_scan: object mismatch
panic: vm_page_dirty: page is invalid!
panic: vmSPACE_fork: entry 0xfffff80019793d00 eflags 50c
```

rename3.sh

Some test scenarios are written or suggested by others. For example, this small rename test scenario by Tor Egge: "Test vulnerability to transient failures when a directory closer to the root directory is renamed". This has triggered multiple deadlocks.

isofs2.sh

This is the latest test and a very simple one. Create an isofs file system with a copy of date(1). Mount the file system and run the copy of "date." This would trigger a "panic: witness_warn" as reported here: <https://people.freebsd.org/~pho/stress/log/isofs2.txt>

marcus5.sh

This is an example of a test that was a spin-off of a search for a different issue. The test triggered a problem with how VFS_SYNC() was implemented: <https://people.freebsd.org/~pho/stress/log/marcus5.txt>

md8.sh

This is a regression test for unmapped unaligned IO over a vnode-backed md(4) volume. So, with the buffer "data" page aligned, the tests are:

```
read(fd, data + 512, MAXPHYS)
write(fd, data + 512, MAXPHYS)
```

The Details

Fetch stress2 by:

```
svnLite checkout svn://svn.freebsd.org/base/user/pho/stress2
cd stress2
make
```

This will build some basic test programs in the sub-directory "testcases." All new development is done in the "misc" directory where there are currently some 400 test scenarios. These tests are often referred to as regression tests. The real value is the way they stress different corners of the kernel. Tests in the "misc" directory can either be run separately or by control of the "all.sh" script. For example, run all the tmpfs(5) scenarios once by:

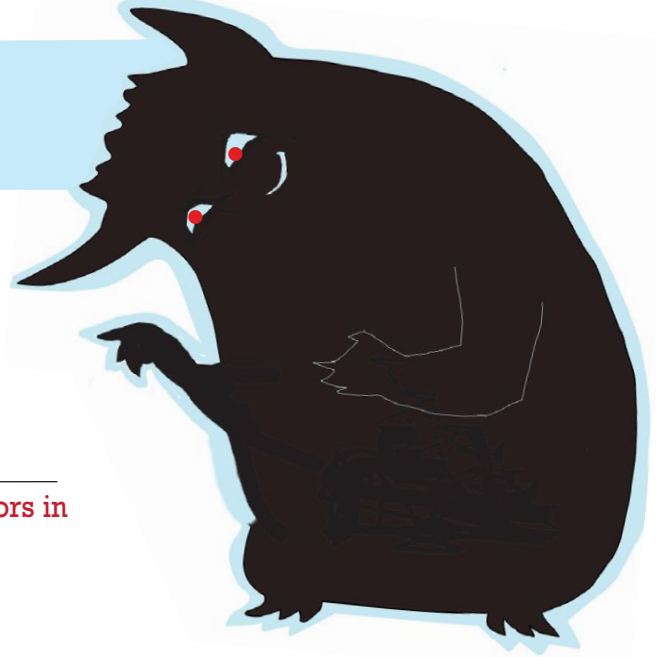
```
$ ./all.sh -o tmpfs*
20150907 22:02:12 all: tmpfs2.sh
20150907 22:06:11 all: tmpfs9.sh
:
```

In Conclusion

Stress2 is a developer tool for finding problems in the FreeBSD kernel. It is not a substitute for real world testing, but just a tool for finding some of the problems.

This work is sponsored by EMC / Isilon Storage Division.

PETER HOLM (pho@FreeBSD.org) has been finding errors in FreeBSD since 1999. The Stress2 test suite is in constant development, as new tests are added as a result of bug reports or patch tests.



ISILON The industry leader in Scale-Out Network Attached Storage (NAS)

Isilon is deeply invested in advancing FreeBSD performance and scalability. We are looking to hire and develop FreeBSD committers for kernel product development and to improve the Open Source Community.



We're Hiring!

With offices around the world, we likely have a job for you! Please visit our website at <http://www.emc.com/careers> or send direct inquiries to karl.augustine@isilon.com.



EMC²

ISILON



The DOs and Don'ts

of File System Benchmarking

BY VASILY TARASOV, ZHEN CAO, MING CHEN, AND EREZ ZADOK

As we know, “everything is a file in Unix”^[14]. Documents, executables, hard disk drives, memory, resource utilization statistics, and even system settings are all accessed and modified through files. As such, the file system is the cornerstone of any Unix derivative and its performance and efficiency are extremely important for overall system speed. Over the years, a myriad of file systems with diverse goals, designs, and implementations have been proposed and developed. File systems that persistently store and retrieve data are, of course, of special importance. The one common element of all file systems is that they all provide an identical (POSIX) API. For example, an application using a POSIX-compliant file system can portably create, open, read, and write a file; make, list, and modify hierarchical directory trees; access and change a file’s metadata; and so on.

Running applications execute file system operations and often need to wait for operations to complete. In many production systems the amount of time that applications spend waiting for file system operations is the main contributor to the total execution time. In this case, the application is said to be *I/O bound* and the best way to improve its performance is by increasing file system speed. But one cannot improve what cannot be accurately measured. File system and storage benchmarking are therefore crucial processes for both evolutionary and disruptive improvements of computer system performance.

In practical scenarios, benchmarking is often used to compare alternatives—for example, when deciding which file server to purchase from several available models or when choosing among different local file systems. Sometimes users need to pick between completely different storage architectures (e.g., local vs. shared storage). With the advent of new storage technologies, users often wonder if upgrading to expensive Flash memory or PCM will improve file system performance enough for their demands. Alas, modern file systems have a large number of configuration parameters that strongly impact their performance [16]; selecting the optimal parameter values can therefore mitigate I/O bottlenecks and eliminate or postpone the need for costly upgrades [4]. To answer these and similar questions, one needs to properly and fairly compare the performance of different systems. Yet this is not a simple task, mainly because so many major and minor details need to be taken into account.

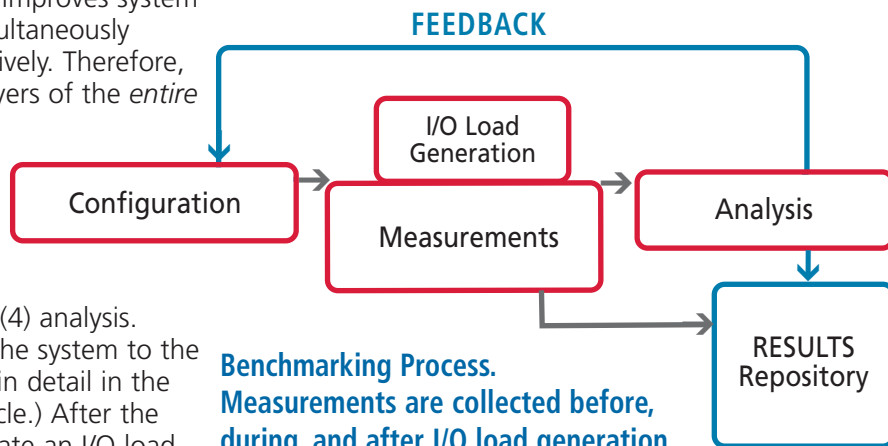
Inexperienced users (and even experts) make mistakes in benchmarking methodology [18, 21]. In this article we describe important principles and techniques to properly assess the performance of file-system-based storage systems. The guidelines presented here are the result of many years of experience with benchmarking file systems and storage in a diverse set of research and engineering projects.

File systems are an integral part of the I/O stack, which consists of at least I/O libraries, VFS layers, file system layers, block layers, and hardware. For a network file system, the networking stack plays an additional, important role in performance. When virtualization is employed, parts of the I/O stack are duplicated inside a VM, effectively doubling the total number of layers [20]; similarly, storage virtualization like RAID, LVM, and stackable file systems add even more layers. Every single layer has a number of alternative configurations: the specific configuration used greatly impacts the overall performance of an I/O-bound application. It is nearly impossible to measure the performance of a file system in isolation, and even if it were possible, measuring a “middleware” layer such as a file system without the rest of the software, OS, and hardware it runs on would not yield meaningful results. The same file system (e.g., FreeBSD’s `ffs` [11]) would definitely exhibit different performance when run in a different environment. Even smaller differences, like the prefetch size at the block layer, can improve (or degrade) performance of a file system. Moreover, due to software and hardware complexity, even if increasing the values of two parameters independently improves system performance, increasing both values simultaneously might impact system performance negatively. Therefore, in practice one always benchmarks all layers of the *entire* I/O stack rather than an individual file system layer.

We divide the benchmarking process into four major steps as depicted in the figure at right:

(1) system configuration, (2) I/O load generation, (3) measurements, and (4) analysis. System configuration brings all parts of the system to the desirable states. (We will cover this step in detail in the System Configuration section of this article.) After the system is configured one needs to generate an I/O load with appropriate characteristics (described in the I/O Load Generation section) and simultaneously start measuring system behavior (explained in the Measurements section). In fact, to capture the initial and final system states, measurements are also taken before and after the I/O load generation. Finally, the results from measurements are analyzed, visualized, and conclusions are made (as detailed in the Analysis section).

Benchmarking is a time-consuming, highly repetitive and iterative process; note the blue *feedback* arrow in the figure. When the goal of an experiment is to compare several alternative configurations or workloads, the whole process is often repeated multiple times for every possible setup. Frequently, the results of the analysis demonstrate unexpected results, and require changes to configurations, workloads, measurements, or all. For example, if during the analysis step file-system memory utilization turns



out to be lower than expected, it might further reveal an inadequately low limit set on the cache size. Then one might decide to repeat the exercise with a higher cache limit. Similarly, intense benchmarking can tickle bugs not seen before, causing system crashes or revealing serious “performance bugs,” which requires code changes (and sometimes design changes), after which the entire set of benchmarks must be rerun. This feedback cycle sometimes has to be repeated many times; not surprisingly, researchers have found the time spent on this benchmarking cycle is far longer than the initial development of the first system prototype.

Before going into the heart of this article we have several important disclaimers. First, benchmarking is a field with a lot of open debates and conflicting opinions. We convey our point of view and hope that this will promote a healthy discussion on how to conduct storage evaluation properly. Second, as with any rules, there are exceptions to benchmarking rules. We try to cover the most common scenarios, but there are plenty of legitimate cases when the rules we present can and even should be changed. Third, we focus on performance, but this is not the only metric by which one decides which file system is “better.” The set of features supported, acquisition cost and total costs of ownership, and power consumption, are just a few examples of other factors that impact the decision on which storage solution to use. Fourth, in real life, experimenters are limited by practical time constraints (e.g., software release cycles, paper deadlines). Therefore, we note both ideal approaches that assume an unlimited amount of time and less perfect, but practical techniques. We believe it is acceptable to deviate somewhat from the ideal methodology provided and that it is well understood which assumptions are made.



System Configuration

System configuration is the very first step of the benchmarking process. Although this step is absolutely crucial for the success of the whole process, it is often unfairly disregarded or considered trivial. Without a detailed understanding of the system’s configuration, it is not possible to properly analyze results, deduce generalized conclusions, or repeat identical or derived experiments at a later time. The three main goals of system configuration are to (1) learn a configuration space, (2) consciously and reliably set system parameters to the required values, and (3) collect exhaustive configuration descriptions for future reference.

Most often the complete configuration space is not known before benchmarking starts. If that is the case, then all configuration parameters and their possible values should be carefully studied, understood, and listed. The configuration space can be roughly divided into two parts: parameters that can be modified only manually (e.g., by installing different network or storage devices), and parameters that are easy to change programmatically (e.g., file system types or `pdflush` frequency). Although some parameters cannot be set programmatically, they can often be read programmatically (e.g., a disk drive’s model is visible in kernel boot logs). It is important to record all parameters even if one expects them to remain unchanged throughout all planned experiments. Benchmarking plans have a tendency to change. In our experience, recording the complete information on the system configuration saves a lot of time when plans inevitably change. In addition, knowing the details of hardware setup significantly facilitates the analysis stage. Specifically, we recommend to note HDD and SSD sizes, models, speeds, and capabilities (e.g., whether the `TRIM` command is supported and what is the parallelism level of an SSD); network card models and speeds; hardware RAID controller models and settings (e.g., read/write cache size, modes); the number and models of CPUs, cores, and their speeds; main memory size and topology; system BIOS settings; and more. Certain software parameters also remain static and cannot be easily changed but should still be recorded (e.g., kernel, distribution, and library versions). Configuration harvesting should be as automated as possible (e.g., using shell scripts).

Parameters that can be modified programmatically should be set to the appropriate values before every experiment. This ensures that between experiments (which sometimes spans multiple weeks), system parameters are not changed accidentally. Many configuration parameters have default values. For example, one often does not need to specify block or inode sizes when formatting a file system. We highly recommend never to trust the defaults because they vary with environments. By assuming that defaults are the same everywhere, one risks comparing two different configurations without even knowing so. Therefore, set all parameter values explicitly. As we mentioned earlier, one must not limit these rules to the file system settings. All I/O stack parameters (e.g., `pdflush` frequency, dirty memory high/low watermarks) should be set explicitly. If a software volume manager or iSCSI is used, then the corresponding settings need to be examined and set as well. We also recommend collecting the list of running processes and verifying that no unexpected processes that potentially disrupt the experiments are running (e.g., `cron` jobs, background daemons).

After setting all parameter values, we recommend reading all of them back. This is an extra safety measure that ensures all the settings actually took place and there were no mistakes in the configuration-setting commands. We find it necessary to read parameters twice per experiment—in the beginning and at the end of the run to ensure that parameters did not change during the experiment. Save the output of all commands and store those configurations along with the results.

Some modern storage systems support advanced features like deduplication, compression, encryption, etc. It is important to be aware of such features. For example, if compression is supported, it will be important at the I/O load generation step to write data with a consciously selected compression ratio. At times one might decide to temporarily disable certain advanced features to simplify the results analysis or to identify a feature's performance cost and efficiency.

In modern infrastructures, hardware is often shared (e.g., a JBOD connected through a SAN or an NFS server). Traffic from other users might disrupt the experiments, causing the results to be sensitive to the environment and hurting the ability to reproduce the results. Whenever possible, try to ensure that nobody else uses shared resources during the experiments. For example, one can disable remote logins to the machines and limit NFS mounts to a set of specific nodes. If it is not possible to technically restrict access to the shared resources, ask other users not to use the resources during your experiments. Although this is not a guaranteed approach, it still increases the chances of undisturbed results.

Professionals often resort to using several identical machines in parallel to speed up experiments. For instance, one can split a set of experiments to execute one part on one machine while the other part runs on another machine. In ideal cases, this allows one to complete experiments twice as fast. Using two machines is not ideal because even seemingly identical machines do not perform exactly the same. In fact, it was shown that the performance of modern HDDs can vary as much as 20% within the same model line [10]. However, because of practical time constraints, it is often necessary to use multiple machines. In this case we recommend spreading the instances of exactly the same experiments across different nodes and then reporting the average values along with a variance metric (e.g., standard deviation). This way the differences between machines are included in the performance numbers (instead of inadvertently contributing to the performance differences between workloads or configurations). Note that the benefit of such an approach is that the results describe system performance not on one specific machine but rather on a population of "identical" machines, which is more general and has a higher value.

Another common approach for speeding up experiments is to artificially limit the RAM size so that smaller datasets can be used while still generating ample I/O activity (more on the dataset size in the section on I/O Generation). In this case both the initial dataset creation and the cache warmup phases run faster. The hidden assumption is that if the RAM-to-dataset ratio remains the same, then performance with larger datasets (and larger RAM) remains the same. Logically this makes sense but we are not aware of any studies that have verified this assumption. Therefore, if time permits, it is better not to limit RAM size artificially and use the complete dataset sizes. In case of a time constraint, we recommend ensuring that after limiting the RAM size, there is still the same number of DRAM slots accessible to every CPU node (relevant in NUMA nodes only).

Before every experiment, the system needs to be configured to exactly the same state as in other experiments in the series—except, of course, the differences dictated by the experiment's objective (e.g., comparing one configuration to another). Configuration steps should be designed so that they bring the system to the same starting point before the actual workload runs. For file system benchmarking, we recommend formatting the file system before every experiment. Remounting the file system is not enough because the history of file system usage in previous experiments can impact performance in the following experiments. After mounting a file system, it is sometimes desirable to age the file system to a realistic (though reproducible) state because file system performance deteriorates over time [1]. If aging cannot be performed (e.g., it takes too long or the tools are not available), then using an empty file system is an alternative. At the very least, we recommend filling the file system with as much data as it expects to store in production because many storage devices perform more slowly as more data is written to them. For instance, HDD throughput is higher on the outer tracks (which are filled first) because they rotate at a higher linear speed. SSDs, on the other hand, use unwritten space for pre-erased blocks, so their performance also decreases with higher utilization. In fact, we recommend overwriting SSDs completely before experiments to trigger garbage collection typical in the long-running production systems. Executing the **TRIM** command on an SSD may not make sense because at best it brings the SSD to the optimally performing state, which usually is not the goal of a realistic evaluation. Worse, however, is that the SSD can be left in a different state before every experiment because **TRIM** can run asynchronously and it is not known when the Flash Translation Layer (FTL) actually gets to erase all blocks.

It is worth noting that modern systems often inherently have randomness in them. For example, some file systems allocate disk blocks randomly to provide consistent performance over the lifetime of a file system [15]. In this case it is practically impossible to bring the system exactly to the same state. Every single configuration procedure will produce a slightly different setup. Such differences, however, are perfectly fine. They either will not impact performance numbers significantly, or if they do, then the user will become aware of system's sensitivity to the initial state (e.g., report higher standard deviations). After that, one can decide if it is an acceptable performance variance for a specific environment.

During configuration and the steps that follow, it is important not to omit any errors returned by the tools or the system. We recommend stopping the experiment immediately if any single command exits with a non-zero status or if there is an error in the kernel or system logs. After the error is fixed or identified as irrelevant, the run should be resumed from scratch.

For distributed file systems there are many servers and clients in the mix and the steps mentioned above need to be repeated by each node.



I/O Load Generation

After successfully configuring the target systems, the next step is to generate the I/O load with the required characteristics, which we call the *I/O workload*. The two most important things that users need to keep in mind during this process are: (1) understanding **what** benchmarks or applications do in as much detail as possible, and (2) thinking carefully about **how** to run the benchmarks or applications.

Below we describe four common methods for generating file system loads—micro-benchmarks, macro-benchmarks, I/O traces, and application-level benchmarks—each applicable to specific scenarios [21].

- **Micro-benchmarks.** These benchmarks are designed to exercise a few (usually one or two) types of file-system operations—e.g., measuring how many creates-per-second a file system can achieve. Such benchmarks are useful when the goal is to measure the impact of a small change in a system, to (later) better understand the results of macro-benchmarks, or when users want to isolate the effects of a specific part of the system. The results of micro-benchmarks are more valuable and meaningful if presented with results from other types of benchmarks. A few examples of the commonly used micro-benchmarks include fio [6], iозone [3], and some of Filebench's personalities [5].
- **Macro-benchmarks.** The goal of macro-benchmarks is to provide an estimate of the system's performance when deployed in production. These benchmarks exercise multiple file system operations and are designed by observing and characterizing real-world workloads and then simulating them. Examples of macro-benchmarks include Filebench's Web/Mail/File personalities [5] and SPEC SFS®.
- **I/O traces.** Storage developers recognized early that in complex cases simple counters are not enough to analyze system behavior, and therefore added the ability to record every operation in the system. I/O traces are a collection of timestamped records about file-system or block-level operations captured on a specific system. After a trace is collected in one system, it can then be replayed on other systems to evaluate their performance. Trace replay can provide an accurate estimate of storage performance, but users still need to ensure that the actual traces used are representative of the real workload. For example, traces should cover extended periods of time to capture as many as possible occurring usage patterns.

There are several methods for replaying I/O traces, which cause debates on which replay method is the most appropriate [21]. Some replay traces with the original timings. However, traces are usually collected on older and slower systems, so the use of original timings does not stress newer and faster storage enough. Another approach is to replay traces as fast as possible ignoring the timings and fixing the total number of outstanding requests. In this case the interdependencies between requests are ignored and the results might differ from reality. Finally, as a middle ground, one can replay the trace with a fixed speedup factor and limit the maximum number of simultaneously outstanding requests. When using trace replay for evaluation, we recommend using all of the above methods and making appropriate conclusions. To the best of our knowledge, there are no widely available file-system trace replayers; the btreplay tool can be used for replaying block-level traces.

- **Application-level benchmarks.** Previously described benchmarks only mimicked real applications. Application-level benchmarks, on the other hand, exercise targeted systems by deploying real applications on them. The benchmarks then simulate the way users operate such applications in real life. For example, the TPC-C benchmark [22] requires the deployment of a real database software which it exercises in a manner representative of complex OLTP application environments, portraying the activity of a real-life wholesale supplier.

I/O workloads observed in real life and produced by benchmarks are often characterized using a set of common metrics, such as operation ratios, I/O size distributions, level of parallelism, sequentiality, and dataset sizes. The premise behind such characterization is that the performance of storage systems depends mainly on macroscopic statistical properties rather than on small details of workloads [19]. For example, the operation ratio is the percentage of each file system's operation type (e.g., read, write, create) in the overall mix. Similar but coarser characterization distinguishes metadata-intensive (high percentage of namespace operations) vs. data-intensive workloads. File system performance is typically fairly sensitive to this characteristic because namespace-management operations are designed and implemented quite differently from the data management. In an evaluation, one usually needs to use the benchmark that produces a workload with characteristics close to the ones observed in the target environment.

One especially important property of a workload is the dataset size. Some workloads can fit their datasets completely in RAM; and then the underlying storage does not impact file system performance. More frequently, however, the dataset size is larger than the size of the file system cache, and, therefore, both memory and storage subsystems are exercised. We recommend setting the dataset size several times larger than the available RAM size. Even better is to experiment with different dataset sizes, keeping all other workload characteristics unchanged. We would like to stress that file system performance can be very sensitive to dataset and RAM sizes. We have shown that a dataset size increase as small as 6MB can result in almost 10 times lower throughput [18].

In many production deployments, workloads do not cause the system to reach its peak performance. In other words, there is a lot of idle time between requests submitted to the file system. When evaluating a storage system one might decide to respect realistic idle time and observe system behavior in a non-stressed scenario (e.g., measure request latencies when the rate of incoming requests is relatively small). Another approach is to observe how the system behaves under the highest load, which allows one to measure system's peak performance. We recommend evaluating systems varying the load from modest to highest levels—and reporting performance of all points along this continuum.

Several nontraditional aspects of workloads have become important in recent years. If a storage system supports deduplication or compression, then benchmarks should generate content with appropriate compression or deduplication levels. Many older benchmarks wrote zeros or arbitrary data to a file system, both of which do not evaluate the system properly. One needs to pick a compression ratio that is expected in a specific environment—e.g., if a target storage system is used for storing documents, then a typical compression ratio is in the 3 to 5 times range [9]. Newer versions of fio, Filebench, and other benchmarks support compression features. DEDISbench [13] is an I/O benchmark for deduplication systems and it includes the distribution of duplicate content as one of its inputs.

The next step in benchmarking is to understand the condition of when to terminate the I/O load. There are two approaches to specify a stopping condition: *time-based* and *job-based*. A time-based approach specifies the fixed duration of the benchmark run. An example of this category is running stress tests to see how many requests a web server can handle during the peak hour of a day. In contrast, job-based approaches specify the amount of work that needs to be completed for each benchmark run. For example, when testing the speed of sorting a 1TB text file in a big data system, a job-based approach is more convenient and makes more sense than a time-based approach. Note that in this case, the running time actually becomes a relevant metric reflecting the performance of the system. In practice, both approaches are valid, and choosing the right one depends on the use case.

While deciding how long an experiment should run, users also need to consider the typical cycles in system operations, such as cache flushes, log wrapping, major and minor compactions, etc. The benchmark should at least cover multiple iterations of the longest operation cycle, so that all modes of the system behavior are evaluated. Moreover, some benchmarking tools have a warmup phase before the actual run of simulated workloads to let the system reach a steady state. During the warmup phase usually no metrics are collected. We consider it unnecessary to treat the warmup phase differently because it is often important to understand what was happening to a system with respect to performance and other metrics during the warmup. So, users should treat the warmup as part of the run, collect all measurements periodically, and later, if needed, users can discard data collected from the warmup stages of the experiments.

After finishing the benchmark run, some recommend executing an *fsync* operation. However, we consider this a superfluous step as it really depends on the specific characteristics of the real workloads. If no *fsync* operations occur at the end of the real workload, there is no need to add an *fsync* after the benchmark. In fact, if the working set size and the duration of the benchmark are sufficiently large, then

dirty data flushes should occur many times during the run and are included in the performance results.

Last but not least, we encourage the avoidance of custom-made storage benchmarks. This complicates any results verification, reproduction, and hurts their trustworthiness.



Measurements

While generating the workload, one needs to collect metrics that both measure system performance and describe system behavior in a broader sense. Measurements serve as an input to the results analysis—the final step of benchmarking. The quantity and the quality of the measurements to a large extent determine the speed and efficiency of results analysis.

The metrology of file system benchmarking asks two main questions: (1) **what** to measure, and (2) **how** to measure. The first question is about what are the important metrics we need to collect to facilitate the following analysis; and the second question is about how to accurately and efficiently measure those relevant metrics. The answers to both questions are not absolute, and depend largely on the storage devices used, file system type, workloads, and the analysis goal. In this section we discuss general guidelines to help answer these questions.

—What to Measure—

The task of identifying relevant metrics can be challenging considering the complexities of modern file and storage systems; knowing the common characteristics of “good” metrics can thus simplify this task. A good metric has four decisive characteristics: (1) informative, (2) well-defined, (3) quantitative, and (4) simple. Because the purpose of metrics is to help engineers and researchers understand the system, each metric should be informative enough to facilitate one or more analysis tasks, including sanity checks, environment monitoring, system behavior analysis, performance evaluation, and troubleshooting. An informative metric is often a clear indicator of system environment (e.g., temperature, network load), resource utilization (e.g., CPU or memory use), system performance (e.g., throughput, latency), or system events (e.g., page faults, interrupts, allocations of new blocks).

A good metric should also be well-defined and without any ambiguities. A well-defined metric should have a clear context under which the metric is collected. The same type of metric can be significantly different in different contexts. For example, in a setup that uses Network File System (NFS), throughput measured at the application level (say, in MB/sec) is not equal to the throughput measured at the NFS client level because of the client-side page cache; nor will application level throughput match the throughput measured at the RPC level because of the client side persistent cache such as FSCache [8]. Furthermore, the throughput measured at the block layer of the NFS server is another completely different metric.

A well-defined metric should also have clear and meaningful boundaries that mark the start and the end of the metric. Many benchmarking studies fail in this regard, for example, by choosing a somewhat arbitrary warmup period and reporting only performance metrics after that warmup. The problem is that a fixed warmup period is not necessarily meaningful: the warmup may be intended to fill a cache, but the degree of cache fullness after the fixed warmup period varies in different workloads. Then the seemingly reasonable warming up will cause the reported results to depend on unknown and probably different initial cache states. As already mentioned in the section on I/O Load Generation, a better alternative is to mark the start of the metric at the very beginning of the experiment (before warmup), and measure until the work is finished.

Being quantitative is another important feature of good metrics. Quantitative metrics tend to be more accurate and reproducible than qualitative metrics; and quantitative metrics also lead to more objective analysis than qualitative ones. As Lord Kelvin—the famous physicist who accurately measured the absolute zero temperature—said, “When you measure what you are speaking about and express it in numbers, you know something about it, but when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.” For example, describing throughput as “bursty” when writing data to an empty page cache is less clear than “a throughput of 1GB/sec drops to 80MB/sec after 5 seconds.”

The fourth characteristic of a good metric is simplicity, which makes the metric easy to measure. A simpler metric also makes the analysis following the measurement more intuitive and less error-prone.


Now that we understand what desirable features to look for when searching for good metrics, we discuss how to identify relevant metrics. Measuring a single performance metric is a bad practice; the metrics we collect for an experiment should be comprehensive enough to conduct a wide range of analysis tasks following the experiment. Therefore, we should collect metrics in all of the following categories:

- **Performance metrics.** These metrics, such as throughput and latency, are direct indicators of performance, and thus the most important. There may be multiple throughput metrics and multiple latency metrics. In NFS, for example, there are ops/sec and MB/sec throughput metrics at different levels of the storage and networking stack; there are several latency metrics ending at different layers as well (e.g., latency upon page-cache-hits, and latency upon FSCache-hits).
- **Resource utilization.** Use of CPU, memory, storage I/O, and networking are also important metrics. They are direct indicators of system efficiency, and are helpful in understanding system behavior and troubleshooting.
- **Metrics of the setup.** We should measure metrics that validate our setup and serve as evaluation baselines. For example, is the caching device indeed much faster than the primary storage device, and what is the maximum possible speed-up with 100% cache-hit-ratio? If possible, we should measure the metrics of our setup by ourselves because the metrics provided by manufacturers are often too simple or too optimistic to be trusted.
- **System-specific metrics.** For example, compression and deduplication ratios reported by the systems that support these features.
- **Metrics of external events.** These metrics, such as the request rate of external clients, are especially important in benchmarking environments we do not fully control.
- **Environment metrics.** Such as temperature of the server machines, and the degree of congestion of the storage networks.

—How to Measure—


After knowing the metrics to measure, the next step is to carry out the measurements. A good measurement should have the following features:



- **Accuracy.** Being accurate means the measured value of a metric is close to the real value of the metric. Accurate measurements form the foundation to make correct decisions.
- **Precision and stability.** Being precise means the results of multiple measurements of the same metric



Rack-mount networking server

Designed for BSD and Linux Systems
Up to **5.5Gbit/s** routing power!

Made for  **FreeBSD**


PERFECT FOR

- ▶ BGP & OSPF routing
- ▶ Firewall & UTM Security Appliances
- ▶ Intrusion Detection & WAF
- ▶ CDN & Web Cache / Proxy
- ▶ E-mail Server & SMTP Filtering
- ▶ Anti-DDoS and clean pipe filtering


KEY FEATURES

- ▶ 6 NICs w/ Intel igb(4) driver w/ bypass
- ▶ Hand-picked server chipsets
- ▶ Netmap Ready (FreeBSD & pfSense)
- ▶ Up to 14 Gigabit expansion ports
- ▶ Up to 4x10GbE SFP+ expansion


I Gbit/s Copper	Ports	Chipset
L800-G808-1	8x Gbe RJ-45 ports	8x Intel i210 AT; PEX8618
L800-G808-2	8x Gbe RJ-45 ports	8x Intel i210 AT; PEX8618
L800-G428-1	4x Gbe RJ-45 ports	1x Intel i350 AM4
L800-G428-2	4x Gbe RJ-45 ports	1x Intel i350 AM4
I Gbit/s SFP (Fiber)	Ports	Chipset
L800-S406-1	4x Gbe SFP ports	i350-AM4
10GbE Copper	Ports	Chipset
L800-T202-1	2x 10GbE RJ-45 ports	Intel X540
L800-T203-1	2x 10GbE RJ-45 ports	Intel X540
10GbE SFP+ (Fiber)	Ports	Chipset
L800-X204-1	2x 10GbE SFP+	Intel 82599ES
L800-X205-1	2x 10GbE SFP+	Intel 82599ES
L800-X405-1	4x 10GbE SFP+	Intel 82599ES; PEX8724




DESIGNED FOR
FreeBSD



DESIGNED FOR
GNU / Linux



DESIGNED FOR
FreeBSD
pfSense



DESIGNED FOR
Sense

Designed. Certified. Supported

contactus@serveru.us | www.serveru.us | 8001 NW 64th St. Miami, FL 33166 | +1 (305) 421-9956

are close to each other. High precision makes us confident in the consistency and reproducibility of the measurements.

- **Efficiency.** Efficiency is important to ensure our measurement does not introduce excessive system loads that skew the system performance.
- **Automation.** Automated measurement is more reproducible and less prone to uncertainties caused by human errors.

We also recommend the following practices when measuring metrics of file and storage systems:

- **Repeat experiments and measurements multiple times.** Multiple runs make us confident of the precision; for metrics that vary a lot, multiple runs, especially when shown as box plots (or CDFs), describe a more factual picture of the range of the metric. For example, a distribution of network latencies, instead of a single latency, would better illustrate the dynamic nature of networking and multi-layer structure of storage.
- **Measure metrics periodically during experiments instead of only once at the end.** Systems are dynamic and metrics often vary during the run. When shown in time series graphs, frequent measurements can capture the dynamics of the metric and filter out random noise. In our experience a period of 10 seconds between measurements provides enough granularity without causing an excessive overhead.
- **Tools like `vmstat`, `iostat`, and `nfsstat` are not the original sources of measurements.** These tools read numbers from the `/proc` file system, perform extra calculations, and then print results to the user. The calculations, however, might not be trivial and in combination with poor labeling may lead to wrong conclusions. One common example is the service time metric in `iostat` that is valid for old HDDs and makes little sense for highly-parallel SSDs [2, 12]. Therefore, when possible, we recommend sticking to the original data sources and performing calculations on your own during the analysis phase.
- **Record common timestamps for every measurement.** These timestamps can help coordinate events from different subsystems (e.g., storage and networking, client and storage server). When the experiment involves multiple machines, it is important to synchronize the time using the Network Time Protocol (NTP).
- **Ensure that the performance penalties of measurements are negligible** by running experiments with and without the measurements.
- **Save measurements along with the configuration and workload descriptions used in the experiment.** It is convenient to make every experiment's folder completely self-sufficient.

→ Analysis

The analysis step either reaches the final goal of the whole evaluation or guides the design of additional experiments (see *Feedback* arrow in the figure on page 15). In this section we share several analysis practices that we have found useful.

- **Formatting.** Measurements are often collected in wildly diverse and not post-processing-friendly formats—e.g., outputs of `iostat` and `vmstat` tools are quite different and cannot be directly fed to Gnuplot or a spreadsheet tool. We find it convenient to first format all measurements to some common structured format. In our experience a CSV format provides the best common ground: it is accepted by most analysis or visualization tools, can be read by humans, and parsed by programs. Files in CSV format are also convenient for sharing with other involved parties who might be using different toolsets. We usually use CSV files where the first column contains the timestamps and the remaining columns have corresponding measurement values.

After running many experiments, an evaluator ends up with many experimental results. Having a consistent file-naming convention and directory structure is highly useful. This not only simplifies navigation through results for humans, but also allows us to easily use the same analysis and visualization scripts for all experiments.

- **Visualization.** With so many numbers collected, it is practically impossible to comprehend them without summarizing the numbers with graphs. We found Gnuplot and Python's matplotlib to be powerful tools that are easy to script; they dramatically accelerate the plotting of many experiments.

We always recommend starting with time series graphs—the graphs where the X-axis shows the time and the Y-axis shows one of the measured metrics. For example, throughput-time series graphs allow us to quickly understand if performance was stable during the whole run. If not (e.g., due to the caching effects in the beginning of the run), when reporting the average throughput, one might consider only the throughput after the cache became warm. Similarly, plotting the memory utilization versus time allows us to confirm the point at which the cache became full. In many experiments one can see periodic dips in performance due to periodic cache flushes or segment compactions.

When preparing graphs that compare several independent experiments, it is important to show the variability of every metric. Specifically, instead of using bar graphs with a single average number per experiment, box plots show more detailed statistical information using the same space (mean vs. median, outliers, upper and lower quartiles, and more).

- **Sanity checks.** One of the first tasks of analysis is to verify that the measurements make general sense. For example, if the throughput at the application layer turns out to be several times higher than the HDD bandwidth (and the dataset does not fit in RAM according to your experimental setup), then something is probably wrong with the experiment. It can be that the dataset is much smaller than expected, data is being completely deduplicated, or something else. Another example is when many metadata operations are observed in an experiment that was designed to be data-intensive. Of special concern should be when the results appear to be “too good to be true” as they are often not. Another task of sanity check is to verify that all logs do not contain error and warning messages.

- **Overhead.** Frequently we see that people refer to degradation in performance as overhead—e.g., if after enabling deduplication, throughput falls from 100MB/sec to 80MB/sec, then it is sometimes said that the overhead of deduplication is 20%. This, however, is not precise. Overhead is about the resource utilization, such as CPU cycles, memory usage, I/O bandwidth, but not about performance [7]. For the above example, I/O bandwidth usage might have increased twofold (e.g., for bringing the deduplication index to RAM), so the actual I/O overhead is 100%. A contrary example is when, after enabling some feature, performance improves by 50%. It is not a good practice to report only the fact that the feature increases performance 1.5 times. In fact, it is quite possible that CPU utilization grew from 25% to 75%, which means that CPU overhead of the feature is 300%. Often the increase in performance is not acceptable if the overhead is too high, so the analysis should report both performance improvements and resource utilization.

- **Performance.** Two basic metrics of file system performance are throughput and latency. Depending on the context, throughput can be defined as IOPS (I/O operations per second) or as MB/sec. We recommend to always start from IOPS because this is a more universal and less ambiguous metric that applies equally well to both metadata and data operations. Depending on the operation mix and I/O

RootBSD

Premier VPS Hosting

RootBSD has multiple datacenter locations,
and offers friendly, knowledgeable support staff.
Starting at just \$20/mo you are granted access to the latest
FreeBSD, full Root Access, and Private Cloud options.



www.rootbsd.net

size, different IOPS can translate differently to MB/sec. For one workload, high IOPS might still mean low MB/sec (metadata operations or random I/O with small I/O sizes), while low IOPS for another workload might translate to high MB/sec (large multi-MB write operations). After the IOPS metric has been analyzed, one can translate it to MB/sec as necessary.

Note that for some software, high throughput is important, whereas for others, low latency has a higher value. Average latency for a single-threaded system (one request is in flight at all times) can be computed as a reciprocal of IOPS. However, most of the storage stack is multi-threaded, which increases throughput manifold when multiple requests are submitted simultaneously. But if too many requests are submitted, then latency can start to grow as a result of queueing. We recommend analyzing how throughput depends on latency—e.g., putting throughput on the X-axis and average latency on the Y-axis allows us to see for which throughput system latency is still acceptable. Monitoring system queues often helps better understand this behavior.

Conclusions

There is a wide variety of storage hardware, software, and workloads. Every time a new system is built or an old one upgraded, decisions need to be made on which storage setup to use. After functionality, performance is the second most important system characteristic. Although it is easy to determine if the functionality offered by the storage layer suffices, it is much harder to ensure that performance is at the required level. Storage benchmarking is the discipline devoted to answering this complex question. Yet people have been benchmarking storage for a long time and we still see a lot of poor practices: unclear configuration, inappropriate workloads, poorly selected metrics, and erroneous analysis. In this article we have presented several tips and techniques that we learned the hard way over the years. We hope the information presented here will be useful to readers and we also hope and encourage the community to use and publish more quality storage performance evaluations in the future. ●

Thank you!

The FreeBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.



The
FreeBSD
FOUNDATION

Are you a fan of FreeBSD? Help us give back to the Project and donate today!
freebsd.foundation.org/donate/

Iridium



Platinum

NETFLIX



Gold

ARM

JUNIPER
NETWORKS

Silver

Limelight
NETWORKS

XINUOS

Netgate

Tarsnap

vmware

ixsystems

Google

Please check out the full list of generous community investors at freebsd.foundation.org/donate/sponsors

AUTHOR BIOS

EREZ ZADOK received his PhD in Computer Science from Columbia University in 2001. He directs the File Systems and Storage Lab (FSL) at the Computer Science Department at Stony Brook University, where he joined the faculty in 2001. His current research interests include file systems and storage, operating systems, energy efficiency, performance and benchmarking, security, and networking. He has received the SUNY Chancellor's Award for Excellence in Teaching, the U.S. National Science Foundation (NSF) CAREER Award, two NetApp Faculty awards, and two IBM Faculty awards.

MING CHEN received his BS and MS in Computer Science from Beihang University in China. He is a fifth-year PhD candidate at Stony Brook University, working with Professor Erez Zadok in the File Systems and Storage Laboratory (FSL). His research interests include analysis, design, and implementation of distributed storage systems and cloud-computing systems.

ZHEN CAO received his BS in Software Engineering from Fudan University, China. He is a third-year PhD candidate at Stony Brook University, working with Professor Erez Zadok in the File Systems and Storage Laboratory (FSL). His interests include benchmarking and auto-tuning complex storage systems.

VASILY TARASOV is a researcher at IBM Almaden Research Center. He received his PhD from Stony Brook University in 2013. His interests include system performance analysis, design and implementation of distributed systems, and efficient I/O stacks for ultra-fast storage devices. He maintains and contributes extensively to the popular Filebench benchmarking framework.

REFERENCES

- [1] N. Agrawal, A. C. Arpaci-Dusseau, and R. Arpaci-Dusseau. Generating realistic impressions for file system benchmarking. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*. (2009)
- [2] M. Brooker. Two Traps in iostat: %util and svctm. (2014)
- [3] D. Capps. IOzone file system benchmark. www.iozone.org.
- [4] E. Zadok, A. Arora, Z. Cao, A. Chaganti, A. Chaudhary, and S. Mandal. Parametric Optimization of Storage Systems. In *HotStorage '15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*. (2015)
- [5] Filebench. <http://filebench.sf.net>.
- [6] fio—flexible I/O tester. <http://freshmeat.net/projects/fio/>.
- [7] G. Heiser. Systems benchmarking crimes. <https://www.cse.unsw.edu.au/gernot/benchmarking-crimes.html>.
- [8] D. Howells. FS-Cache: A Network File System Caching Facility. In *Proceedings of the 2006 Linux Symposium*, volume 2. (2006)
- [9] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and Performance Evaluation of Lossless File Data Compression on Server Systems. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*. (2009)
- [10] E. Krevat, J. Tucek, and G. R. Ganger. Disks are like snowflakes: No two are alike. In *Proceedings of HotOS XIII: The 13th USENIX Workshop on Hot Topics in Operating Systems*. (2011)
- [11] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197. (August 1984)
- [12] B. Mildren. Monitoring IO performance using iostat & pt-diskstats. (2013)
- [13] J. Paulo, P. Reis, J. Pereira, and A. Sousa. DEDISbench: A Benchmark for Deduplicated Storage Systems. In *Proceedings of the International Symposium on Secure Virtual Infrastructures (DOA-SVI)*. (2012)
- [14] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375. (1974)
- [15] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*. (2002)
- [16] P. Sehgal, V. Tarasov, and E. Zadok. Optimizing Energy and Performance for Server-Class File System Workloads. *ACM Transactions on Storage (TOS)*, 6(3). (September 2010)
- [17] SPEC SFS@ 2014. <https://www.spec.org/sfs2014/>.
- [18] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of HotOS XIII: The 13th USENIX Workshop on Hot Topics in Operating Systems*. (2011)
- [19] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*. (2012)
- [20] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok. Virtual machine workloads: The case for new benchmarks for NAS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. (2013)
- [21] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80. (May 2008)
- [22] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification. www.tpc.org/tpcc. (2004)

THE INTERNET NEEDS YOU

GET CERTIFIED AND GET IN THERE!

Go to the next level with



Getting the most out of
BSD operating systems requires a
serious level of knowledge
and expertise ● ● ● ● ● ● ● ●

SHOW YOUR STUFF!

Your commitment and
dedication to achieving the
BSD ASSOCIATE CERTIFICATION
can bring you to the
attention of companies
that need your skills.

NEED AN EDGE?

● **BSD Certification can
make all the difference.**

Today's Internet is complex.
Companies need individuals with
proven skills to work on some of
the most advanced systems on
the Net. With BSD Certification

**YOU'LL HAVE
WHAT IT TAKES!**

BSDCERTIFICATION.ORG

Providing psychometrically valid, globally affordable exams in BSD Systems Administration

FreeBSD™ Journal
Foundation

is published by The FreeBSD

PORTSreport

by Frederic Culot •

THE LEVEL OF ACTIVITY during the November–December 2015 period was especially high on the problem-fixing front. During this period, we were pleased to welcome miwi@ back and to extend ian@’s commit bit to the ports area. Also, some major ports were updated, which may require caution when upgrading, as described below.

IMPORTANT PORTS UPDATES

As always, many exp-runs were run by Antoine@ (21 in total), to check whether major ports updates are safe or not. Among those important updates we can mention the following highlights:

- CMake updated to 3.4.1
- ruby-gens updated to 2.5.0
- setuptools updated to 19.2
- freetype2 updated to 2.6.2

Other important changes include the update to ncurses 6.0, which requires the rebuild of all ports using it in case you do not use binary packages. Also, a change in the tor startup script now makes it necessary to manually set up the logging. As usual, more information is available in the `/usr/ports/UPDATING` file, which should be read carefully before upgrading major ports.

NEW PORTS COMMITTERS AND SAFEKEEPING

In November we were pleased to restore miwi@’s commit bit since he now has time to work on ports again, and in December we were delighted to extend ian@’s commit bit to the ports tree, so that he will be able to update his u-boot ports more easily.

Some commit bits were also taken in for safekeeping, after more than 18 months of inactivity (jhay@, max@, sumikawa@, alexey@, and sperber@). Sadly, nox@’s commit bit was removed, as he passed away. For more on Juergen Lock’s contributions to FreeBSD, you can read a short summary on our [memoriam page \(https://www.freebsd.org/doc/en/articles/contributors/contribdevelinmemoriam.html\)](https://www.freebsd.org/doc/en/articles/contributors/contribdevelinmemoriam.html).

STATISTICS

A few statistics for the November–December period: around 110 committers applied 4,187 changes to the ports tree, which is a little less than the previous period. However, the activity on the bug-fixing front increased significantly, with 1,217 problem reports closed, representing an increase of more than 10% compared to the September–October 2015 period. Thanks to all of you who reported issues and to all committers who applied the fixes!

tips

Here are a few tips related to ports updates. First, you need to identify the outdated packages to be upgraded. The following command could be used to do so: `pkg version -vl'<'`

Then we must find out whether there is any specific processing that needs to be followed for the packages that were previously identified. Using the following command, you can get a list of the impacted packages that need special care: `grep -B1 AFFECTS /usr/ports/UPDATING`

Lastly, `pkg upgrade` could be used to update binary packages, and any port having non-default options should be recompiled.

In order to be warned whenever ports need updates, you can add the following command to the crontab, so that portsnap (see <https://www.freebsd.org/doc/handbook/ports-using.html>) updates the ports tree daily and an email is sent that contains all outdated packages.

```
0 1 * * * root portsnap cron update && pkg version -vl'<'
```

Having completed his PhD in Computer Modeling applied to Physics, FREDERIC CULOT has worked in the IT industry for the past 10 years. During his spare time he studies business and management and just completed an MBA. Frederic joined FreeBSD in 2010 as a ports committer, and since then has made around 2,000 commits, mentored six new committers, and now assumes the responsibilities of portmgr-secretary.

svn UPDATE

by Steven Kreuzer

It's the start of a brand new year and we have already seen several exciting updates that highlight FreeBSD's continued commitment to high-performance networking. In addition, FreeBSD is quickly becoming a mature and powerful platform to use in your virtualization projects, be it as a host running on bare metal or as a guest running somewhere in the cloud. While you can look forward to upgrading your production machines to 10.3-RELEASE in the very near future, you should pay very close attention to HEAD because 2016 is going to be a very exciting year for 11-CURRENT.

New sendfile(2) with Support for asynchronous I/O

One of the most exciting changes to hit HEAD has been the addition of an implementation of the sendfile(2) system call. The result of an ongoing development partnership between NGINX and Netflix, the new sendfile significantly speeds up large TCP data transfers by adding support for asynchronous I/O. Even more impressive is that the new sendfile is a drop-in replacement, so it will not be necessary for you to make any changes to your applications to take advantage of these improvements. You can now expect significantly better performance in cases where the old sendfile was blocked on disk I/O. (<https://svnweb.freebsd.org/changeset/base/293439>)

Support for Netmap in bhyve Guests

While bhyve continues to see very active development and is quickly gaining widespread adoption, a recent commit will help address one of the biggest disadvantages that plague all virtualization deployments. Network I/O virtualization tends to exhibit poor perform-

ance, especially under heavy load, and until recently it was a non-trivial task to eliminate these bottlenecks. Guest operating systems that are running under bhyve now have the ability to make use of netmap, a framework for high speed packet I/O, to achieve near native performance. (<https://svnweb.freebsd.org/changeset/base/293459>)

EC2 Enhanced Networking Enabled by Default

Also known as SR-IOV (Single Root I/O Virtualization), this extension to the PCIe specification allows devices such as network adaptors to appear as multiple separate physical devices to the hypervisor or guest operating system. SR-IOV enables network traffic to bypass the virtualization stack to achieve network performance that is nearly the same as in non-virtualized environments. SR-IOV is now enabled by default when building EC2 images for Amazon Web Services. (<https://svnweb.freebsd.org/changeset/base/293739>)

LLDB Enabled by Default on amd64 and arm64

LLDB is a next-generation, high-performance debugger, which is made available under a BSD-style license. After extensive testing has shown that it works as well as the in-tree gdb version, it has been promoted to the default debugger on both amd64 and arm64. LLDB also provides some level of support for FreeBSD on arm, mips, i386, and powerpc, but is not yet ready to replace gdb as the default debugger on these platforms. (<https://svnweb.freebsd.org/changeset/base/292350>)

ZFS Boot Environments

If a system is booted with ZFS, a new menu item will appear in the loader that will contain an autogenerated list of ZFS boot environments making it easier to switch to an alternate root file system. This will quickly become a handy feature if you would like to switch between different versions of FreeBSD or need to recover from a failed upgrade. (<https://svnweb.freebsd.org/changeset/base/293001>) ZFS boot environment is

also available in the UEFI loader as well. (<https://svnweb.freebsd.org/changeset/base/294073>)

UEFI Gains Terminal Emulation Support

Based on the existing vidconsole implementation, it is now possible to emulate a video terminal in UEFI. (<https://svnweb.freebsd.org/changeset/base/293233>) Shortly after that change was introduced, the Beastie menu was added to the UEFI console. (<https://svnweb.freebsd.org/changeset/base/293234>)

Updates to head/contrib

The base FreeBSD userland is made up of quite a few utilities, some of which are developed outside of the project. In the past few months we've seen quite a few updates to the third-party software that helps create a great user experience.

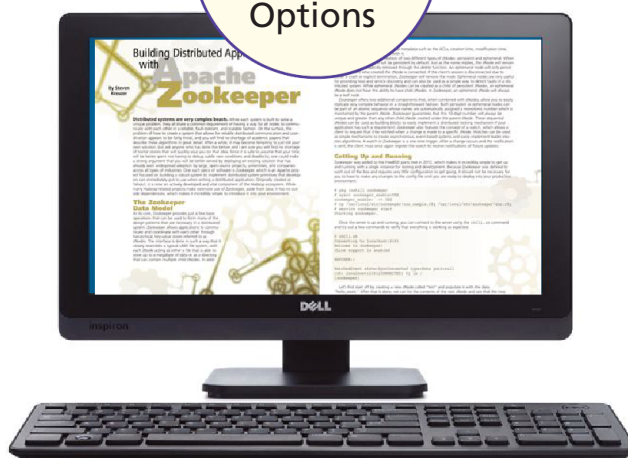
- clang and LLVM have been upgraded to version 3.7.1 (<https://svnweb.freebsd.org/changeset/base/292735>).
- less has been updated to version v481 (<https://svnweb.freebsd.org/changeset/base/293190>).
- ntp has been updated to version 4.2.8p5 (<https://svnweb.freebsd.org/changeset/base/293423>).
- bmake has been upgraded to version 20151220 (<https://svnweb.freebsd.org/changeset/base/292733>).
- OpenBSM has been upgraded to version 1.2a4 (<https://svnweb.freebsd.org/changeset/base/292432>).
- Unbound has been upgraded to version 1.5.7 (<https://svnweb.freebsd.org/changeset/base/292206>).●

STEVEN KRUEZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.



The Browser-based Edition (DE) is now available for viewing as a PDF with printable option.

NEW
Viewing & Printable Options



The Browser-based DE format offers subscribers the same features as the App, but permits viewing the *Journal* through your favorite browser. Unlike the Apps, you can view the DE as PDF pages and print them.

To order a subscription, or get back issues, and other Foundation interests, go to www.freebsd.foundation.org

\$19.99
YEAR SUB
\$6.99
SINGLE COPY

The DE, like the App, is an individual product. You will get an email notification each time an issue is released.



this month

In FreeBSD

BY DRU LAVIGNE



Over the next few issues, we'll be taking a closer look at some of the new features making their way into the 2016 releases and the developers behind those features.

This month we chat with **ALLAN JUDE**. Allan has been a very busy contributor since receiving his doc commit bit in 2014 and his src commit bit in 2015. Several of his projects will make it into this year's 10.3 and 11 releases.

Q Tell us a bit about yourself. How did you get started with FreeBSD and what is your involvement with the FreeBSD Project?

A I started as a FreeBSD user back in 1999 when I wanted to run my own IRC server. After downloading the software from the Internet, I found there was no .exe in the archive. After asking around some, I was told that I would need a "shell account" to be able to run the software. I found a provider that would rent me such a shell account for around \$10/month. Now-a-days you can rent a virtual machine with about four times as much RAM and even more CPU for about the same price, even with 16+ years of inflation.

I was fairly comfortable with the command line, but had never used anything other than DOS and Windows, so there was a lot to learn. As I learned more and more, I eventually stopped renting shell accounts and started selling them. In 2002 I started my own business renting IRC servers and shell accounts, and quickly became quite good at securing my servers against malicious users. As my knowledge expanded, I started also doing consulting work, setting up web-hosting stacks like Apache/MySQL/PHP for people, custom compiled from source.

When I went to college, I was lucky enough to go to a school that actually taught Linux and BSD together. There I met some other BSD users and first heard about this wonderful thing called BSDCan. Sadly, rather than going to my first BSDCan in 2006, I didn't make it to one until 2012. When I finally did attend, it changed everything. I didn't get all that much out of my first BSDCan, other than the excitement and energy that was in the air, and meeting a few people I had only heard about on mailing lists. I

decided that not only would I come back next year, but I'd submit a talk, and share some of my BSD sysadmin experience with the audience. Why wait for next year? So I submitted the proposal to EuroBSDCon 2012 in Poland, and was accepted. Giving my first talk was quite an experience, but it was well received, and the audience asked insightful questions. I've been to almost every BSD conference I could manage since then.

In 2013, on the way to BSDCan, I decided that the documentation in the FreeBSD Handbook on ZFS was lacking a lot of detail, so I started writing. I attended my first doc sprint, and got up to speed on how to write documentation for the project. In August of 2013, a fan of my sysadmin podcast, TechSNAP.tv, asked me about starting a BSD-centric podcast. At first I didn't think there would be enough news to sustain a weekly show, but after the viewer, TJ, who became our first producer, and I began writing up overviews of a few weeks' shows, I quickly learned otherwise. After a short search, Kris Moore was tapped to be my cohost. Then, a few months later, on my flight to EuroBSDCon 2013 in Malta, I started looking at what could be done to make the FreeBSD installer handle ZFS better. My first draft was sloppy and full of errors, but an especially helpful committer worked with me to get it into shape. Before long, it was part of the operating system. BSDCan 2014 included a big surprise, being granted my doc commit bit. Shortly after BSDCan 2015, I was granted my src commit bit.

Now, I use FreeBSD every day at my day job, ScaleEngine.com, a video streaming company, except on Wednesdays, when I do the BSDNow.tv podcast with Kris Moore. Then my evenings are spent hacking on diverse projects,

from documentation, to standardizing configuration files across utilities, to improving the command line interface of ZFS, and adding features to the FreeBSD bootcode and loader.

Q For some time, you have been working with UCL. What is UCL exactly and how does it compare with other configuration languages? What advantages does it provide over traditional configuration files?

A UCL, or the Universal Configuration Language, was designed by another FreeBSD Developer, Vsevolod Stakhov <cebka@freebsd.org>, for his own utility, rspamd. It consists of a library, libucl, that can parse the config files into objects that the application can use to configure itself, and which can emit modified config files back out. The idea is to have a config file that is easy for a human to read and write, while at the same time being able to manipulate it programmatically—a config format that any experienced sysadmin or user will understand and appreciate.

The UCL config syntax itself is based on the syntax of NGINX and Bind, but slightly modified in a way to make it also compatible with a loose interpretation of JSON:

```
category {
    key = value;
}

othercategory {
    subcategory {
        max_size = 10kb
        expiration = 3d
        array1 = [1, 2, 3]
        array2 = [
            thing1,
            thing2,
            thing3,
        ]
    }
}
```

UCL syntax has a number of features that make it easier for humans to write. It does not require the semicolon terminator at the end of a line. It supports arrays, but unlike JSON, allows the last item in a list to be followed by a comma, to reduce the diff as items are added to the list. There is also “syntax sugar,” where values can contain units, like k (1000), kb (1024), h (hours), and d (days). Booleans can be specified as any of: true/false, on/off, or yes/no. UCL also supports comments, in all three popular styles, including single line (//) and multi-line (/* ... */) C comments, and the

standard hash (#) character that many config file formats use.

The power comes from the native support for variables, macros, and includes. Variables are set by the application, for the user to use in their configuration file, like \$HOST for the system hostname. Macros allow the application that is interpreting the configuration file to extend the configuration language with additional functionality. The includes system allows additional files to be included into the configuration. Includes supports priorities, so the application or user can control which value is used if a setting is defined in two different bits of configuration. Support for GLOB patterns and search paths allow multiple files to be included, such as /usr/local/etc/appname/*.conf. There is also support for remote includes, with optional signature verification.

libucl itself can interpret UCL, JSON, YAML, and Msgpack. It can also output the configuration in all four of these formats. An application that uses libucl to parse its config file, will accept any of these; as to the application, they all look the same.

Q FreeBSD is transitioning to using UCL. Can you describe what users can expect in the upcoming 10.3 and 11.0 releases of FreeBSD?

A Time constraints and compatibility concerns mean that users of 10.3 will not see any changes. However, starting with 11.0, a number of config files will change format. For compatibility and ease of transition, all of the tools will continue to accept their original file formats, but will gain support for the new UCL-based config files. One of the first things I implemented as part of the UCL conversion was a version identifier in the config file that will make future transitions easier, and allow applications to detect when they are given a new config file. Users who upgrade an existing machine to 11.0 will have the option to just continue using their existing config files, or try to convert them. Users who install a fresh 11.0 system will have the new UCL config files by default. The only complication here is documentation: since the old format is still supported, the documentation will need to be retained, if for no other reason than to be a resource to those who are converting their configuration files.

At this time, there is no plan for tools to automatically convert old config files to the new format, but there is nothing stopping someone from adding one to the ports tree.

My initial targets for conversion are: newsyslog.conf, login.conf, jail.conf, pw.conf, and wpa_supplicant.conf.

There are a lot of config files to cover, so I am always interested in hearing from people who want

to help, or even just comment on which config files they would like to see converted next.

I would also like to revive my bhyveucl (github.com/allanjude/bhyveucl) project. Originally it was a shell script that read a UCL config file and wrote the complex bhyve command line to launch the VM as described, and configure the network as required. It was sidelined when I joined a project to implement the config file parsing directly in bhyve itself. That project was put on indefinite hold at the request of the bhyve authors, as they are working on a number of enhancements, including USB sup-

“At this time, there is no plan for tools to automatically convert old config files to the new format, But there is **nothing stopping someone from adding to the ports tree.**”

—ALLAN JUDE

port, a plugin architecture for networking and storage, and a host of other things. These will require a much more expressive config file than I had originally designed. However, that work is taking longer than expected, so there may be value in reviving bhyveucl. Even once the work in bhyve is completed, the configuration file may be rather complex, and a utility like bhyveucl that can take a simpler config file and convert it into the more expressive “machine description” that bhyve will require may still be of great value.

Q Improvements are also being made to bsdinstall to provide support for ZFS boot environments. For readers unfamiliar with this feature, can you provide an overview of its benefits? What type of work was needed to add this support and when will it be available to users?

A ZFS boot environments are a way to have multiple root (/) file systems and switch between them at reboot in order to revert a problematic upgrade, or to dual boot multiple versions of the operating system. They can be managed manually, but there is a utility in ports, sysutil/beadm, that provides a nice user interface. Each boot environment is a ZFS dataset, although in most cases, it is a ZFS clone of the existing file system, so takes no additional space until changes are made. If you clone your / file system before an upgrade, and the upgrade does not work as expected, just use 'beadm activate oldbootenv' and reboot, and your system will

boot from the clone of / from before the upgrade. Other file systems, like users' home directories, are not affected.

The initial support for ZFS boot environments was introduced to the FreeBSD installer in FreeBSD 10.0. When you use the 'Automatic root-on-ZFS' mode in the installer, it creates pool-name/ROOT/default, which will be your first boot environment.

There are two ways to use boot environments. PCBSD creates a new environment before each upgrade, starts a jail chrooted in that environment, and does the upgrade there, then reboots into that environment. I personally prefer to just use the 'default' boot environment, and create clones of that before each upgrade, in case I need to roll back. Another key factor is deciding what should be included in the boot environment, and what should remain untouched when switching between them. The base operating system (/bin, /sbin, /usr/bin, /usr/sbin, and /etc) is usually included, but it can depend on your environment if you want /usr/local (where applications installed with pkg are put) to be unique to each environment, or stable across them all.

The problem with boot environments as they shipped in 10.0–10.2 is that if you end up with one that doesn't boot correctly, the only way to switch is by manually manipulating the loader prompt, or booting from a live CD/USB and switching the 'active' boot environment.

I have been working on an additional menu option in the beastie loader menu that allows you to select a different boot environment. This allows you to quickly, easily, and safely switch between the different root datasets at boot time. What sets this apart from the way it is currently done in PCBSD with GRUB, or in IllumOS, is that rather than reading the list of boot environments from a configuration file, the list is generated by examining the ZFS pool itself, so the list is always up to date.

The other complication is encryption. If the user GELI encrypts their ZFS pool, they currently require a UFS partition, or a second not-encrypted ZFS pool to sort the kernel on, so that support for GELI encryption could be loaded to decrypt their file system. In both cases, it is not possible to support boot environments, because the kernel does not reside on the boot where the environments were created.

I have been working on solving this issue by implementing support for GELI decryption in the bootcode and loader. I hope that this work will also make it into 10.3. This will allow booting

om a single encrypted ZFS pool, offering full support for boot environments, even in the presence of encryption.

Q You have written extensively on ZFS, including the ZFS chapter of the FreeBSD Handbook and the ZFS Mastery series of books with Michael W Lucas. How did you get interested in ZFS and what benefits has it provided to you and your business?

A I first got interested in ZFS when my company needed to store large quantities of video files, in a flexible and safe way. I had little experience with enterprise storage, having never used anything more complex than motherboard BIOS-assisted mirroring on any of my machines. I found the concept of pooled storage and copy-on-write to be very interesting and to fit our needs quite well. The administration and configuration interface for ZFS was also extremely easy and powerful, and I quickly became very comfortable with it. The more I learned about ZFS, the more I wanted to share it with everyone.

I have learned a lot since setting up that first ZFS server in 2011. At our company each customer gets their own ZFS dataset, so we can more easily move customers between servers, manage their snapshots independently, and assign quotas. When I first deployed ZFS, we just had one big dataset for video, and each customer had a directory. The problem was that if a customer purged a large number of their videos, or cancelled their account, we did not regain that space until the snapshots were destroyed. We did not want to lose the snapshots of our other customers' data, while at the same time being able to get that space back, so we have transitioned to using a separate dataset for each customer. We also use the ZFS space accounting system to bill our customers.

The biggest advantage to our business has been the flexibility, resiliency, and tunability of ZFS. With the tools provided by FreeBSD and ZFS, not to mention DTrace, we can easily monitor and diagnose any performance problems. Having the details of each disk directly exposed to the operating system, rather than hidden behind a RAID controller has also saved time and made managing failing disks easier.

Q The BSD Now podcast is into its third year. Have you seen any trends in BSD usage or perception over that time?

A When the idea for the show was first proposed, I didn't think it would work. After we worked on it for a while and actually started it, I was pretty confident in it. The biggest surprise to

me has been that neither Kris nor I have burned out yet, and still enjoy doing the show each week. I think a big part of that is the very positive feedback we get from the community. My personal favorite part of the show is the interviews.

I have definitely felt an uptick in the perception and adoption of BSD since the launch of the show. We get feedback every week about someone new trying out, or switching over to, a BSD. My concern is the retention ratio, how many of them stick with it.

I wonder what things the projects can do to help newcomers get over those initial hurdles, and become lifetime users like I have. I think the biggest complaint we see is hardware support on newer laptops, and I hope that is something that can be addressed going forward.

Q What other projects are you working on or plan to start in the near future?

A In addition to more work on implementing UCL config files in FreeBSD, and my command line utility for working with UCL called `uclcmd` (github.com/allanjude/uclcmd), I have a number of ideas for new ZFS features. Along with a system to track changes to the ZFS command line interface, for use by scripts that automate parts of ZFS, I am also working on implementing the additional hashing algorithms that were recently added to ZFS to the FreeBSD kernel, so they can be used on ZFS. One of them, SHA512t256, which is a SHA512 truncated to 256 bits since that is the maximum size of the checksum in ZFS, is approximately 50% faster than a regular SHA256 when calculated on 64bit x86 hardware. I have also been working on converting utilities in FreeBSD to `libxo`, a library that makes the utilities able to output JSON and XML, in addition to the regular text output.

My slightly loftier goal is a project called Zoro, a successor to `sysutil/zxfer`, to manage ZFS replication. Zoro would manage snapshot creation and retention, bookmarks, and replication. Some of the ideas I have for it will be best implemented with a little help from the ZFS side, so it might lead to some new ZFS features as well.

I have my fingers in a number of different areas of the system, either trying to make my own job easier, allow FreeBSD to reach more of the potential I see in it, or solving problems that others have, in hopes of increasing adoption of FreeBSD and ZFS. •

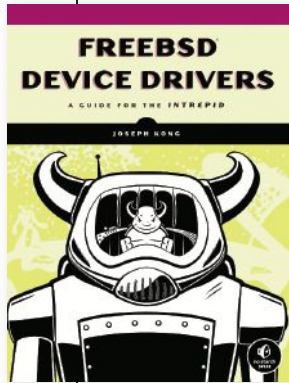
Dru Lavigne is a Director of the FreeBSD Foundation and Chair of the BSD Certification Group.

BOOKreview

by Simon Gerraty

FreeBSD Device Drivers

A Guide for the Intrepid by Joseph Kong



PublisherNo Starch Press, May 2012
 Print List Price.....\$49.95
 eBook List Price\$24.99
 ISBN.....978-1-59327-204-3
 Pages.....352

Device drivers are the code that allows an operating system to deal with hardware. They differ from most of the code in the kernel in that they are often maintained by people more familiar with the hardware than any given operating system.

FreeBSD Device Drivers covers the infrastructure within FreeBSD that is important to device drivers. As such, it should be very useful to those wishing to write or adapt drivers for FreeBSD.

The author makes it clear in his introduction that the reader is assumed to have some familiarity with operating system internals, and he even provides suggested prerequisite reading. In keeping with its intended audience, the book presents the services the kernel provides to device drivers in terms of their APIs only. There is not very much discussion of what happens behind those APIs except where it helps to make sense of them.

The content is arranged in a reasonable progression, starting from the basic scaffolding that every device driver needs as well as resource allocation and thread synchronization. The auto-configuration process by which drivers claim hardware they recognize is covered, as is the Common Access Method (CAM) used for storage devices, DMA, USB, and finally network adapters and their interactions with the network stack.

There are plenty of code listings with commentary to explain what is going on. There are also some simple examples in the early sections on pseudo devices (things that look and smell like a device, but have no associated hardware), but most of the listings are from real device drivers. The format is perhaps a little terse for those reading for general interest, but should generally be appreciated by the intended audience. I thought it worked quite well.

Since many drivers have code to do the same things, the later chapters skipped over those that had effectively been fully described earlier. I appreciated that. I hate the way many technical books unnecessarily pad themselves with redundant information—the worst culprits being those that give you say a hundred pages of useful info, and then three or four hundred pages of verbatim man pages!

I think I fit within the intended audience for this book. I've studied the internals of several operating systems, have ported and tweaked a few device drivers, but have had little to do with hardware since the CP/M days and the Z80. I can at least attest to the fact that transistors do not work after you let the smoke out!

The chapters I found most interesting were those that dealt with the strictly device driver-related APIs. There were a couple of sections where I thought a bit more depth might have helped. There are lots of DMA-related APIs, and obviously few drivers use them all, but seeing some real examples might have been useful.

USB is weird (probably designed by a committee) and something I knew very little about, but was interested to learn. The `ulpt` driver covered in this chapter is a decent example, but USB can be used for a broad range of devices, and perhaps a second example might have been useful.

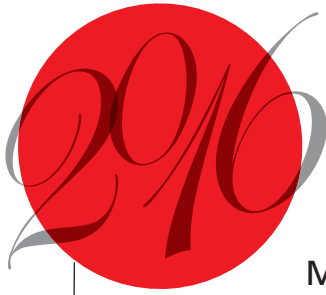
All in all, though, I think the book strikes a pretty good balance for its stated purpose. Its mostly “just the facts ma’am” style works well, but it provides plenty of references for those wanting more detail or background.

This is definitely worth a read if device drivers in FreeBSD are something you need or want to know about. ●

SIMON GERRATY is a Distinguished Engineer at Juniper Networks where he has worked on Junos (FreeBSD-derived OS) since 2000. He is a NetBSD and FreeBSD committer, and has maintained `bmake` since 1993. He has used various UNIX systems (PCs to mainframes) since about 1984.

THROUGH APRIL 2016

BY DRU LAVIGNE



Events Calendar

The following BSD-related conferences will take place in March and April 2016.

More information about these events, as well as local user group meetings, can be found at www.bsdevents.org.

AsiaBSDCon • March 10–13 • Tokyo, Japan



<https://2016.asiabsdcon.org/> • This is the annual BSD technical conference for users and developers on BSD-based systems. It provides several days of workshops, presentations, a Developer Summit, and an opportunity to take the BSDA certification exam in either English or Japanese. Registration is required for this event.

Hackathon • April 22–24 • Essen, Germany



<https://wiki.freebsd.org/DevSummit/201604> • The second annual Hackathon and DevSummit will be held at the LinuxHotel in Essen. This event is open to both FreeBSD committers and contributors. There is a nominal charge for food and lodging.

LinuxFest NorthWest • April 23 & 24 • Bellingham, WA



<http://linuxfestnorthwest.org/2016> • This is the 17th year for this annual, community-based conference. This event is free to attend and always has at least one BSD booth in the expo area and at least one BSD-related presentation.

SUBSCRIBE TODAY



PEOPLE ARE TALKING ABOUT freeBSDTM JOURNAL

AVAILABLE AT YOUR FAVORITE APP STORE NOW



Go to www.freebsd.foundation.org
1 yr. \$19.99/Single copies \$6.99 ea.



**Testers, Systems Administrators,
Authors, Advocates, and of course
Programmers** *to join any of our diverse teams.*

**COME JOIN THE
PROJECT THAT MAKES
THE INTERNET GO!**

★ DOWNLOAD OUR SOFTWARE ★

<http://www.freebsd.org/where.html>

★ JOIN OUR MAILING LISTS ★

<http://www.freebsd.org/community/maillinglists.html?>

★ ATTEND A CONFERENCE ★

